AN EXECUTION MODEL FOR HARDWARE/SOFTWARE COMPILATION AND ITS SYSTEM-LEVEL REALIZATION

Holger Lange and Andreas Koch

Tech. Univ. Darmstadt
Embedded Systems and Applications Group (ESA)
Darmstadt, Germany
{lange,koch}@esa.cs.tu-darmstadt.de

ABSTRACT

We introduce a new execution model for orchestrating the interaction between the conventional processor and the reconfigurable compute unit in adaptive computer systems. We then characterize the architectural and OS-level requirements of implementing the model, and demonstrate how they can be achieved on a real hardware platform running under a full scale multi-tasking virtual protected memory operating system. Experimental measurements show the efficiency of our solution, and also prove that reconfigurable computing can be competitive with processors even for non-streaming, pointer-chasing applications.

1. INTRODUCTION

Reconfigurable devices (RD) used as compute units (RCU) have improved the execution speed and/or efficiency (e.g., power consumption, integration density) in many application areas (a recent survey can be found in [1]). However, exploiting the potential of the technology usually requires from the developer experience in both computer architecture and digital logic design as well as proficiency in hardware description languages.

For limited application domains (e.g., signal processing) tool flows supporting algorithms description in domain-specific languages, such as MATLAB or Simulink, are now becoming sufficiently mature for industrial use [2] [3]. However, the automatic mapping of programs written in common high-level programming languages (HLL, such as C or Java) to an RCU is still the subject of intense research. Even when tools accept a traditional HLL, they often impose restrictions on specific language features (e.g., no pointers, no conditionals in loops [4][5]) or require additional user-specified annotations to guide the translation (e.g., the nature of data streams [6]). Compilation will fail when a prohibited language construct is used or insufficient annotations are present.

As an alternative, some flows [7] [8] [9] [10] target combinations of a conventional processor (CPU) working in tandem with an RCU. Such architectures are sometimes called *adaptive computers* (ACS).

Here, scarce RD logic capacity can be saved by leaving non-critical or RD-unsuitable sections of the application, such as low-ILP, highly irregular control, floating-point, on the CPU. Furthermore, the software-programmable CPU can act as a fallback for the compiler if it is unable to process some parts of the program for HW acceleration (e.g., due to area limitations). Instead of just aborting the translation, these parts can be compiled to software, and the user informed of the specific difficulties. The program always remains executable and can thus be migrated incrementally to full ACS exploitation, with the user rewriting the problematical program sections as necessary.

In all cases, the compiler requires the interaction between RCU hardware structures on the RD, the software on the CPU, and the entire system architecture (encompassing RD, CPU, memory, etc.) to be orchestrated using a common set of rules, the *execution model*. In contrast to manual hardware design, where one can freely mix-and-match different computing paradigms as well as modify them for special cases, an automatic compiler only has a limited set of techniques available to realize the input algorithms. The execution model thus defines the possible solution space for compiler-generated implementations.

This work first describes the novel execution model used by our compiler Comrade [10] and discusses the hardware / software requirements it imposes on the target environment. The following sections then describe actual realizations of these requirements on a real hardware platform, which will be experimentally evaluated afterwards.

2. EXECUTION MODEL

All of the execution models shown here aim at the compilation of C for the combination of CPU and hardware accelerator (HA), the latter either in the form of an RCU [8] [9] [10], or a dedicated hardware block [7].

The three approaches differ in the granularity of the HW

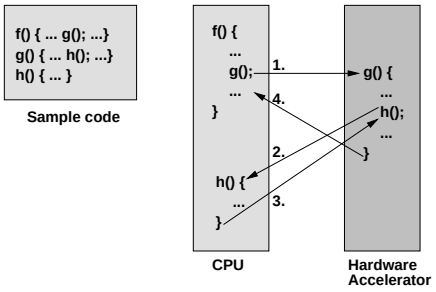

Fig. 1. ASH execution model

```
...
u = (int) sqrt(a + b);
v = c - d;
for (n=0; n<1000; ++n, p=p->next) {
  v += u;
  if (v > 10000) {
    printf("warning: v too large, rescaling");
    v *= 0.271844;
  }
  p->val = v;
}
w = 53 * v;
...
```

Fig. 2. Sample program with HA-unsuitable statements

/ SW partitioning. ASH [7] strictly adheres to the procedure boundaries present in the original C source code. It can thus only move entire C functions between CPU and HA. However, it is able to call SW functions from the HA (see Figure 1). While procedures are indeed natural partitioning boundaries, the presence of C constructs implementable on the HA only with difficulty (system calls, floating point operations) leads to *entire* procedures being ineligible for acceleration, even if the problematical operations occur only rarely (if ever) during actual program execution.

GarpCC [8], Nimble [9] and Comrade [10] use a finer-grained model of interaction that allows HW/SW switches even *within* a procedure. In contrast to ASH, the underlying HW/SW partitioning (not discussed here) is based on actual dynamic profiling data. Thus, slowdowns due to excessive HW/SW communication can be avoided despite the finer partitioning granularity (see experimental results in [10]).

Figure 2 shows a code fragment containing a typical HW kernel (the loop) surrounded by low-ILP code better left on the CPU. For this example, we assume that library functions and floating-point operations are not efficient on the HA (the printf() and especially the sqrt() computation, which is executed just once).

GarpCC, Nimble and Comrade can move just the loop to the HA, and leave the initial and trailing low-ILP parts on the CPU. They differ in their handling of the HA-unsuitable code *within* the HW kernel. All three recognize the exceptional condition $v > 10000$ (which presumably occurs sufficiently rarely in the dynamic profiles that moving the loop to the HA is still profitable) and execute it on the CPU, transfering the live variables (shown as edge labels in Figures 3 and 4) from HA registers to their corresponding SW variables under control of the CPU (the HA is acting in *slave-mode* here). However, GarpCC and Nimble then execute the *entire* remainder of the current loop iteration in SW. Only when the start of the loop is reached again, is the decision made whether to run the next iteration in HW or continue executing in SW.

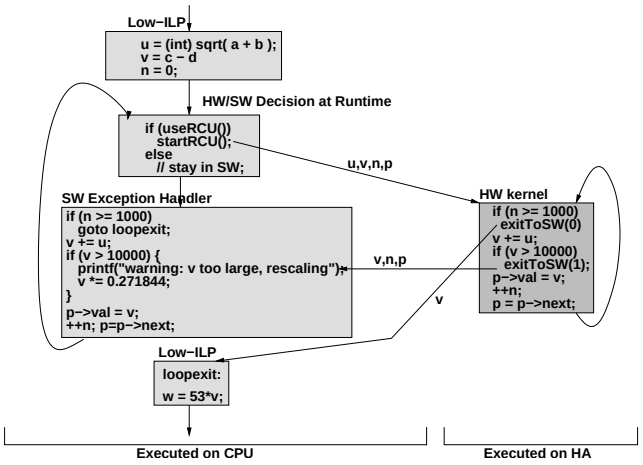

Fig. 3. Example in the Nimble execution model

Our new Comrade model of execution, shown in Figure 4, has even finer interaction granularity. Here, the HA-unsuitable library call and the floating point multiplication execute on the CPU as a *SW Service*. After completion, the SW Service *directly* returns to the HW kernel on the HA. As before, live variables need to be exchanged between CPU and HA. However, note that in the Comrade model, *fewer* variables need to be exchanged due to limiting the scope of the SW Service to just the HA-unsuitable operations (and not an entire loop iteration). In this fashion, a single HW

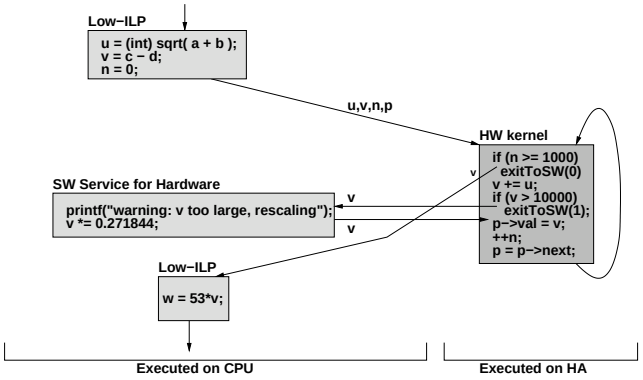

Fig. 4. Example in the Comrade execution model

kernel can access multiple different SW Services, each with just enough code for the requested function.

In addition to the CPU/HA interaction, the execution models need to take the pointer-heavy nature of many C programs into account. With some parts of the programs executing on the CPU, and others moved to the HA, it is crucial that pointer-based data structures (such as the lists used in the example, trees, etc.) can be freely exchanged between the cooperative HA and CPU processing. Furthermore, when operating on pointers, the HA must be able to quickly access memory on its own without intervention of the CPU (called *master-mode*).

3. PLATFORM REQUIREMENTS

In the following discussion, we will number all requirements imposed by the Comrade model as **R***n* to link them with their implementation in later sections of this work.

All of these execution models profit from low-latency communication between the CPU and the HA (**R1**, but not necessarily high-throughput, since only few data items need to be exchanged in a good partitioning). The handling of pointers as discussed in the previous Section can be achieved in a number of ways: Ideally, main memory and possibly parts of the cache hierarchy are shared in a common address space (**R2**, to allow the exchange of pointers) between CPU and HA, and allow high-throughput master-mode access (**R3**). However, many real ACS platforms lack this capability, and all data to be processed by the HA has to be allocated explicitly in dedicated HA-connected memory using a custom function such as malloc_sram(), requiring additional implementation effort (the programmer has to manually partition the memory between the different banks). Furthermore, the latter approach is only feasible for dynamically allocated memory. It fails completely for pre-initialized and stack-allocated data, both of which can occur in C programs. Each of these variables has to be manually copied into the HA-accessible memories, making sure that all contained pointers also point within the HA-memories.

Shared memory spaces are easily realized and handled when using a flat, homogeneous memory without protection between CPU and HA (common for many embedded real-time OSs): When the HA has full access to system memory, it can easily access all C-initialized and stack-allocated data.

This proves more difficult when running the ACS under an OS with virtual and protected memory such as Linux, which is becoming more and more popular even in the embedded world [11]. To integrate the HA seamlessly into such an environment, a number of additional issues needs to be addressed. For security, the HA should only access the memory of the process running the SW part of the application, other memory spaces may not be compromised (**R4**). Furthermore, to maximize the effect of protected memory, the protections should also be applied *within* the HA-accelerated process (**R5**), e.g., making the SW machine code inaccessible to the HA, preserving it from potential corruption). The performance of the rest of the multi-tasking system (including other user processes!) should not be impeded by the operation of the HA. Specifically, the HA may not deny them acccess to memory when they are scheduled by the OS (**R6**). This also extends to not slowing down the SW part of a hardware-accelerated application when sharing memory with the HA (**R7**).

Prior work has considered only a subset of these or other requirements. For example, [12] focusses on ease-of-use of CPU/HA communications by automatically mapping the HA registers to named files in the kernel /proc file system. While this removes the need for memory-mapping the hardware registers, the added file-system overhead increases latencies significantly and would violate **R1** of our model. The work in [13] has a similar approach, but goes further by mapping the *entire* device structure at the configuration level (CLBs, BRAMs, etc.). While this allows the exchange of large chunks of data (by allowing the CPU direct access to the RD on-chip memories), the file-system integration also leads to latencies inacceptable for our execution model. None of these approaches considers the other requirements **R2** to **R7**. [14] proposed a message-passing interface. This would allow the HA master-mode access to memory (**R2**), and the underlying network-on-chip could be extended to provide **R4**...**R6** (none of which is addressed in the paper). The technique also does not deal with **R3** and **R7** at all.

The following sections will discuss in detail how these requirements can be met on currently available hardware, specifically using as HA an FPGA-based RCU, and experimentally evaluate the impact of various design choices.

4. TARGET PLATFORM

Since the simulation of an entire system comprising one or more CPUs, HAs, memories, and I/O peripherals is both difficult and often inaccurate, we employ an actual HW platform to evaluate the practical impact of implementing our execution model.

The Xilinx ML310 [16] is an embedded system development platform which resembles a standard PC main board. In contrast to a standard PC, the CPU and the usual Northbridge ASIC have been replaced by a V2P FPGA [18], which comprises two PowerPC 405 processor cores that may be clocked at up to 300MHz. They are embedded in an array of reconfigurable logic. Thus, the "heart" of the compute system (CPUs, HAs, buses, memory interface) is now reconfigurable and amenable for architectural experimentation. With sufficient care, this rSoC can implement even complex designs with a clock frequency of 100 MHz (a third of the frequency of the embedded CPU).

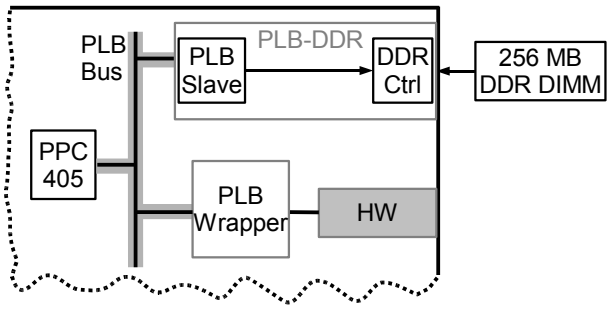

Fig. 5. HA integration via PLB

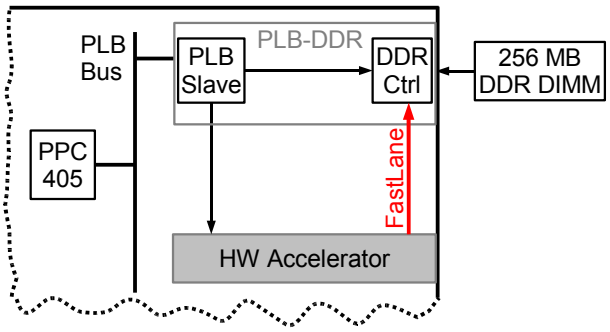

Fig. 6. FastLane: Attaching HA directly to DDR controller

The ML310 is shipped with a reference design, which consists of several on-chip peripherals, attached to a single PowerPC core by CoreConnect [15] buses. These peripherals comprise memory controllers (DDR DRAM and BlockRAM), I/O (PCI-Bridge, UART, the System ACE compact flash based-boot controller, GPIO, etc.), an interrupt controller, and bridges between the different CoreConnect buses.

Note that our architectural concepts as well as their implementation are *not* specific to the ML310, but apply to all platforms with similar characteristics (potentially using more recent RDs, such as the Virtex 4FX and 5FX chips).

5. HIGH-PERFORMANCE HA MEMORY ACCESS

The Xilinx EDK [17] SoC composition tool flow supports two standard means for integrating HAs into the reference design, the faster of which is the PLB attachment (shown in Figure 5).

PLB has features aiming for high-performance operation, but is rather complex. Hence, bus wrappers are nearly always needed when connecting blocks such as HAs to the bus. PLB operates on 64 bits of data at 100 MHz, with two bus wrappers necessary between HA and DDR-DRAM controller for the main memory. Note that the controller itself also requires a wrapper.

As described in [20], the combination of both PLB and Xilinx implementation restrictions renders the memory subsystem insufficient for 64 bit, DDR-200 operation (1600 MB/s theoretical peak performance, the maximum supported by the DDR DRAM used on the ML310). However, since we are experimenting with a *reconfigurable* SoC, we can choose an alternate architecture. To fulfill **R3**, we designed and implemented a new approach to interface the CPU, HAs and the main memory.

The main concept behind the *FastLane* high performance memory interface (described in greater detail in [20]) is the *direct* connection of the memory-intensive HA cores to the central memory controller without an intervening PLB. By also using a specialized, light-weight protocol, we can avoid the arbitration as well as the protocol overhead associated with PLB. This leads to a greatly reduced latency, with no wrapper left between HA core and RAM controller, as opposed to two wrappers in the Xilinx reference design. We can now also make the full bandwidth of the RAM controller available to the HA, eventually enabling true 64 bit double data-rate operation. Figure 6 shows the new memory subsystem layout.

The master-mode side of the HA is connected via FastLane directly to the interface of the DDR controller, but can accept data transfers from the CPU (e.g., live variables for quick HW/SW execution switches, **R1**) via the shared PLB slave. Thus, no additional chip area is wasted on wrappers (which have now become redundant). Both interfaces internally use a simple double handshake protocol, streamlined for low latency and fast burst transfers.

System stability issues (e.g., buffer over/underruns on bus master devices or late IRQ responses, cf. [20]) require that the CPU and other bus master devices always have *priority* over the HA block (which can be explicitly designed to tolerate access delays). The required arbitration logic is completely hidden from the HA within the FastLane interface. The CPU (and other bus master devices) may interrupt master accesses of the HA at any time, while the HA cannot interrupt the CPU, and has to wait for the completion of a CPU-initiated transfer. In this fashion, scheduling decisions by the OS scheduler are enforced at the hardware architecture level, the HA can never let the CPU starve from lack of memory access (**R6**).

6. OPERATING SYSTEM INTEGRATION

An adaptive computing *system* has to consider both HW and SW architectures, since, in the end, it is software applications that are to profit from hardware acceleration. Thus, the HAs must be integrated efficiently and securely with the operating system, the software environment shared by all programs running on the ACS.

With the HA capability to independently access main

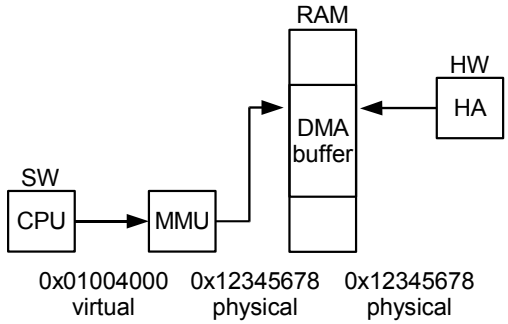

Fig. 7. HW and SW addressing of memory

memory in master-mode, this is non-trivial in an OS environment supporting virtual memory. The memory management unit (MMU, see Figure 7) translates the virtual *user space* addresses as seen by SW applications into physical bus addresses, which are sent out from the CPU via the PLB. Address translations and the resolution of page faults are transparent for SW. Since the HAs do not have access to the MMU (integrated in the CPU)with its page address remapping tables, this implies that hard- and software communication in a virtual memory environment must use *both* virtual user space and physical addresses. Furthermore, since the HA is neither aware of virtual addresses, nor can it handle page faults, the memory pages accessed by the HA *must* be present in RAM before starting the HA.

6.1. Initial Approach: In-Memory DMA Buffer

The straightforward solution to this requirement would be a so-called *Direct Memory Access Buffer* (DMA Buffer). In the Linux virtual memory environment, a DMA Buffer is guaranteed to consist of contiguous physical memory pages that are locked down and always present in physical RAM, they can never be swapped out to disk. As described previously, there are now *two* addresses pointing to the Buffer, the first being the physical bus address as seen by the HA, the second being the virtual userspace address representing the same memory area for application SW. In the example given in Figure 7, a SW program has allocated a DMA Buffer and passes its physical address to the HA. The SW can access this Buffer via userspace address 0x01004000, which is translated to physical address 0x12345678 by the MMU. The HA directly uses this physical address to access the same DMA Buffer.

The fast transfer of live variables between CPU and HA, listed as **R1** of the execution model, is achieved by the CPU issuing reads and writes to the memory-mapped HA registers. These are actually handled by the PLB slave shared with the memory controller, and forwarded to the HA. From the SW perspective, the memory mapped registers are simply accessed via a pointer to a suitable data structure. Bulk data (e.g., image pixmaps, audio stream frames, etc.) is also prepared by the SW within the previously allocated DMA Buffer, which can be manipulated by SW as any other dynamically allocated memory block (e.g., be used in file or network I/O). For HA execution, the offsets of individual data structures within the Buffer are transferred to the HA, which can then proceed to fetch, process, and modify the data under its own control (**R3**). With FastLane (Section 5), this can be achieved in efficient long bursts, fully exploiting the physical memory bandwidth.

6.2. Limitations of Initial Approach

While already integrating the HA with the OS, and efficiently sharing main memory between SW and HA, the above approach of relying on just a dedicated DMA Buffer has disadvantages. A severe one with regard to **R2** is the need to explicitly use two *different* addresses for the same memory location. This renders it impossible to share memory pointers between HW and SW, precluding a broad range of applications from execution on HAs. Furthermore, C-initialized data, typically kept in a program's static data segments (.data for non-zero initialized data, .bss for zero-initialized data), or dynamically allocated on stack or heap, have to be explicitly copied to and from the DMA Buffer every time before and after HA execution. These copy operations take a significant amount of time when transfering larger amounts of bulk data (violating **R3**).

Another problem is that Linux DMA Buffers are generally *non-cacheable* memory areas. This allows the interaction with other master-mode devices (disk or network controllers): Since the CPU cannot be sure that the DMA Buffer has not been written to "behind its back" (leading to stale CPU cache data), it avoids the inconsistency by not using a cache to access the DMA Buffers at all. With the normal use of relatively small ($\approx$ 64KB) DMA Buffers employed for specific purposes (transfering disk blocks or network frames), this strategy is indeed feasible. It fails completely, however, in our ACS execution model: Here, we want to share a potentially large block of general-purpose memory between CPU and HA. If we marked it as non-cacheable, *all* accesses by the CPU would be significantly slowed down (32b single transfers instead of 64b bursts), violating **R7**.

6.3. Refined Solution

As a solution to both problems, we propose to generally keep all data areas (stack, heap and data segments) of a SW executable *inside* the DMA Buffer at runtime (eliminating the time-consuming copy operations). Another benefit of this approach is, that pointers are now *freely interchangeable between HW and SW* (**R2**): The address of every memory

location as seen by the SW differs only by a *constant* offset from the address of the same location seen by the HW. This offset can be transparently removed within the HA address compute path, enabling the HW to use the *same* addresses as the SW. In contrast to conventional approaches, which rely on *explicit* communication between HW and SW to transfer data, our solution allows implicit communication: The native data structures of a program are directly shared by both HW and SW without the need to copy anything, or declare explicit HW data structures or memory areas.

To achieve this, several modifications have to be applied to the Linux kernel. The arrangement of the various areas (instructions, data, heap, stack, etc.) of a new process is established when loading the executable file from disk. Normally, when loading such a program in the common Executable and Linking Format (ELF, [19]) into memory (Figure 8, left), the instructions (in the .text segment and the data segments are laid out starting from the virtual address 0x10000000. No data is actually transferred from disk at this time, only a mapping is established from virtual addresses to the underlying disk file. Only when a virtual address is actually accessed will data be demand-paged in from disk, analogously to handling a virtual memory page fault. The same technique applies to shared library files required by the program (these are mapped-in *below* the program itself). Runtime-managed memory areas such as the heap and the stack, which have no correspondence in the program file, are mapped to anonymous memory: The heap growing upward from the end of the program, the stack growing downward from virtual address 0x80000000.

For loading hardware-accelerated processes in the new Comrade execution model, we have devised the Accelerator-Integrating Shared Layout for Executables (AISLE), shown on the right side of Figure 8. It combines program loading with the management of a DMA Buffer (here set to 16MB) and deviates from the standard layout in a number of ways: First, we move the executable code *outside* of the DMA Buffer by means of a linker script. This has the effect of protecting the CPU code from rogue HA accesses (**R5**). Also, since we aim to use as small a DMA Buffer as suitable for the application (thus reducing the required address width in the HA), we also conserve Buffer space in this manner. The kernel ELF loader was then altered to directly load the data segments of AISLE programs (and only those, see below) into the DMA Buffer, which is mapped-in from 0x10000000 to 0x10FFFFFF. Specifically, we modified the function do_mmap_pgoff to directly load the data from the file into the Buffer, since the HA cannot use the MMU-assisted demand paging. Furthermore, do_brk, which extends the address space of a process for dynamically managed heap memory, was changed to hand-out DMA Buffer instead of anonymous memory (which would be inaccessible to the HA). Finally, we altered setup_arg_pages to

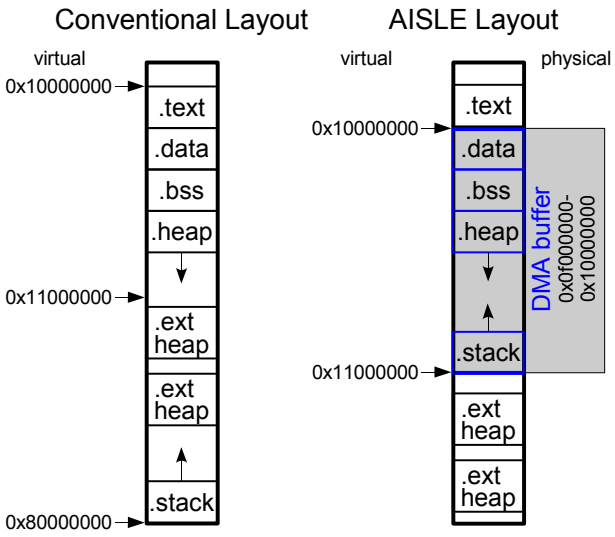

Fig. 8. Conventional and HA-compatible program layouts

initialize the user-space stack of the newly created process within the DMA Buffer, starting at the top and growing downwards. Note that all of these modifications become active *only* when loading an AISLE executable (marked by a flag in the ELF header). Conventional programs are loaded normally, fully profiting from the demand-paging mechanism.

On the hardware side, HA-internal addresses (here: 24b wide) are extended to the 32b supported by the rest of the ML310 by prepending the fixed offset of the DMA Buffer in the 32b memory space (which here requires an 8b prefix). Thus, even an erroneous HA cannot affect other processes (**R4**) or even the code of its own process (**R5**), accesses can occur only with the Buffer.

Since many CPUs used in embedded systems (such as the PPC405 on the VirtexIIpro chip) do not have bus-snooping logic or cache coherency bus protocols (MESI, MOESI, etc.), we rely on SW to maintain coherency between the CPU cache and the (possibly HA-modified) DMA Buffer. Before starting the HA, the control API invalidates and flushes all dirty cache lines located in the DMA Buffer out to actual memory, and invalidates all clean ones also located in the Buffer. Cache lines outside of the DMA Buffer are not affected. Once control returns to the SW process, either after the HA finishes execution or requests a SW Service, all CPU accesses to the DMA Buffer retrieve fresh data. Thus, the SW process can operate at full speed with caches enabled (**R7**).

In this fashion, our AISLE-enhanced OS lets the ACS fulfill the requirements of the Comrade execution model. All of these capabilities are fully transparent to SW developers: C programs need neither explicit copy nor HA-specific memory management calls.

However, in some cases, a greater degree of control can

be beneficial. As an example, a HA-accelerated program might profit from the allocation of large I/O data buffers that do not need to be HA-accessible themselves. While such large buffers could, of course, be realized in AISLE by simply configuring a DMA Buffer of sufficient size, this would be wasteful from a number of perspectives. First, we would require more address bits in the HA, even though the large I/O buffers would never be accessed. Second, since the DMA Buffer requires actual physical memory, that memory would be removed from the demand-paging virtual memory mechanisms, possibly impeding system performance as a whole. To support even these use-cases efficiently, we do provide *optional* API calls that can allow the SW process to request *HA-inaccessible* heap memory outside of the DMA Buffer (the blocks thus allocated are shown as .ext heap in Figure 8).

7. EVALUATION

To demonstrate the effectiveness of the *FastLane* approach, we exercised several system load scenarios. The basic setup is identical in all cases: We mimic the actions of an actual HA by a hardware block that simply repeatedly copies a 2 MB buffer from one memory location to another as quickly as possible, totaling to 4 MB of reads and writes per turn, and measure the transfer rate in MB/s. In addition to the HA, we run a suite of different software programs, chosen for their specific load characteristics, on the CPU. Then, we measure the HW execution time (the time it takes to copy memory data at full speed) and the SW execution time (the time it takes for a given program to execute on the CPU) for both the original vendor-provided as well as our FastLane memory interface. The extreme cases (HA and CPU idle) are also considered.

The first set of measurements shown in Table 1 considers the memory throughput of the HA under different CPU load scenarios. We show the time for a single 2 MB block copy (four mega-transfers) and the resulting memory throughput, both when using the original vendor-provided PLB interface as well as our FastLane attachment for the HA. It is obvious that FastLane significantly increases the HA memory throughput in *all* load scenarios (**R3**), in some cases by a factor of up to 4.3.

The set of measurements shown in Table 2 quantifies the influence of the different memory attachments on SW execution time for programs with different memory access patterns. The results show that, despite its high throughput to the HA, FastLane does not significantly impair the CPU (**R6**): SW execution times are almost unaffected by the HA memory transfer, owing to the absolute priority of the CPU (cf. Section 5) over the HA. In contrast, the original vendor-provided reference design exhibits a steep SW performance decline, increasing execution times by a factor of up to 14x over that of SW running with the FastLane-attached HA. FastLane thus enables the HA to access memory bandwidth that appears to be completely unused by the original memory interface. More detailed discussions are available in [20].

CPU SW Load	V2P ref design Exec Time [ms]	V2P ref design Mem Rate [MB/s]	FastLane Exec Time [ms]	FastLane Mem Rate [MB/s]
idle sys	18.81	213	5.67	705
scp	55.11	73	12.82	312
netcat	53.07	75	19.27	208
gcc	32.14	124	17.42	230
GSM	19.05	210	6.35	630
imgpipe	44.67	90	19.33	207

Table 1. HA runtimes and available throughput using original and FastLane memory subsystem implementations

CPU SW Load	HA inactive [ms]	V2P ref design [ms]	FastLane [ms]
scp	4831	61052	5828
netcat	3130	55938	3901
gcc	40686	166655	52908
GSM	25981	40045	27767
imgpipe	3545	5109	4018

Table 2. SW run times on idle system and using HA attached by original and FastLane memory subsystem implementations

To demonstrate the direct interchangeability of data between HW and SW (**R2**) while maintaining the advantages of the FastLane memory system, we evaluated a pointer chasing application on both HW and SW: The application traverses a randomly linked list of 128 K elements, with each list element consisting of a 32b integer value and a pointer to the next element. On each element, the following operation is performed: If the value is odd, it is increased by one, otherwise it is kept as is. Note that such a trivial operation was chosen on purpose: This test is specifically intended to exercise the capability of the HA to *traverse* irregular pointer-based data structures, instead of relying on the data streaming so common to other ACS applications. Thus, we actively avoid accelerating *computation* (which would bias the results towards the HA).

The results of three implementations for this application are shown in Table 3. First, we evaluated a pure SW implementation. Second, a HA-"accelerated" implementation using just a DMA Buffer, which required copying the list from the normal SW-allocated stack memory into the Buffer. We made sure to use the correct alignment between the SW-allocated memory and the Buffer, otherwise the code would also have to relocate every single pointer within the list (which can be avoided in this manner), and would be even slower. Finally, we evaluated the combina-

SW	HA with DMA/copy	HA with DMA/AISLE
68.3 ms	220.6 ms	27.7 ms

Table 3. Runtimes of the pointer-handling application

tion of DMA Buffer with the AISLE program layout.

The results show that the 100 MHz HA under DMA / AISLE outperforms the 300 MHz CPU by a factor of roughly 2.5, and is even 8 times faster than the HA using the conventional DMA / copy approach. Again, the potential of accelerating the algorithm itself has not even been exploited here.

8. CONCLUSION AND FUTURE WORK

We have introduced a novel model of execution, orchestrating the interaction between a conventional software programmable processor and hardware accelerators. Next, we have shown how to efficiently realize this model on actual hardware.

Furthermore, we have also demonstrated the potential of ACSs for accelerating non-streaming, pointer-chasing code over SW versions. One speed advantage of modern CPUs is often due to the tight integration of fast multi-megabyte caches within the processor, something generally not possible with commercially available RDs. However, the moment the size of irregular data sets exceeds the cache size (such as for railway routing graphs [21]), CPU performance drops to the speed of the memory system. Modern RDs *are* already reaching these speeds, but beyond that can then exploit increased parallelism, both with regard to number of memory banks and processing elements.

The techniques presented so far also did not consider the capability to dynamically reconfigure an RCU, which is now becoming sufficiently reliable to be supported in industrial design flows. Work has already started to support it both in the model as well as the hardware prototype. Other possible areas of future research include the simultaneous sharing of the RCU between multiple *different* SW processes. Also, a further reduction of CPU-HA communications latency is achievable by routing even these requests, which are already optimized by the Comrade execution model, on dedicated connections separate from the system bus. On our experimental platform, this could be achievable by attaching the slave-mode HA to the fast DSOCM (data-side on-chip memory port) port instead of a PLB wrapper.

9. REFERENCES

[1] Gokhale M., Graham P.S., "Reconfigurable Computing", Springer, 2005

[2] Synplicity Inc., "Synplify DSP", **http://www.synplicity.com/products/synplifydsp/index.html**, 2007

[3] Xilinx Inc., "System Generator for DSP", **http://www.xilinx.com/ise/optional_prod/system_generator.htm**, 2007

[4] Gupta S., Gupta R., et al., "SPARK", Kluwer, 2004

[5] Najjar W., Böhm W, et al., "From Algorithms to Hardware", *IEEE Computer*, 08/2005

[6] Gokhale M. B., Stone J. M., et al., "Stream-oriented FPGA Computing in the Streams-C High-Level Language", *Proc. IEEE Symp. on FCCM*, 2000

[7] Budiu M., Venkatarami G., et al., "Spatial Computation", *Proc. Intl. ACM Conf. on ASPLOS*, 2004

[8] Callahan T., Hauser J., Wawrzynek J., "The Garp Architecture and C Compiler", *IEEE Computer*, 04/2000

[9] MacMillen D., "Nimble Compiler Environment for Agile Hardware", Storming Media LLC (USA), 2001

[10] Kasprzyk N., Koch A., "High-Level-Language Compilation for Reconfigurable Computers", *Proc. Intl. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC)*, 2005.

[11] Balacco S., "Linux in the Embedded Systems Market (Vol. VII)", Venture Development Corp, 2007

[12] H. So, A. Tkachenko, and R. Brodersen, "A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH", Proc. 16th Int. Conf. on Field Programmable Logic and Applications (FPL), Madrid, 2006

[13] A. Donlin, P. Lysaght, B. Blodget, and G. Troeger, "A Virtual File System for Dynamically Reconfigurable FPGAs", Proc. 14th Int. Conf. on Field Programmable Logic and Applications (FPL), Antwerp, 2004

[14] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an Operating System for a Heterogeneous Reconfigurable SoC", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), Nice, 2003

[15] IBM, "The CoreConnect Bus Architecture", *White Paper*, 1999

[16] Xilinx, "ML310 User Guide" *(UG068)*, 2005

[17] Xilinx, "Embedded System Tools Reference Manual" *(UG111)*, 2006

[18] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet" *(DS083)*, 2005

[19] Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, TIS Committee, 1995

[20] H. Lange, A. Koch, "Design and System Level Evaluation of a High Performance Memory System for reconfigurable SoC Platforms", Proc. HiPEAC Workshop on Reconfigurable Computing, Ghent, 2007

[21] Müller-Hannemann M., Schnee M., "Finding All Attractive Train Connections by Multi-Criteria Pareto Search", *Proc. 4th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS)*, 11/2004